RV: A Runtime Verification Framework for Monitoring, Prediction and Mining

Patrick Meredith
Grigore Rosu

University of Illinois at Urbana-Champaign (UIUC)
Runtime Verification, Inc.

(joint work with Dongyun Jin and Choonghwan Lee at UIUC)

Assurances for Self-Adaptive Systems

- Self-adaptive systems come with new challenges
  - Increased complexity
  - Reasoning in the presence of uncertainty
  - System analysis pushed to a whole new dimension
    - Concerned not only with functional correctness of the system, but also with how system changes/adapts
  - …
- This talk focuses on a very limited related aspect
  - How to specify a subset of essentially dynamic properties and how to efficiently runtime verify and validate them

Runtime Verification (RV)

- Observe the system while it executes, possibly after appropriate instrumentation
- Check the observed behavior against desirable explicit or implicit properties
- React to detected violations
  - Report error , if used before deployment
  - Recover (when possible), during deployment

RV Workshop/Conference

- Runtime verification had a hard time initially
  - Not testing, not verification; what it is?
- Started in 2001 as workshop, about 20 people
  - ENTCS proceedings
  - Selected papers in journal (FMSD)
- Turned into a conference in 2010
  - LNCS proceedings
  - FMSD special issue
  - 80 participants
- RV'11 had ~80 submissions

Current Limitations of Runtime Verification

- Runtime and memory overhead: system observation and monitoring may add significant runtime and memory overhead
- Limited coverage: if done naively, runtime verification guarantees lack of bugs only in the observed path, but not in other paths
- Specifications: the difficulty of producing property specifications is underestimated. Developers do not want to write them.

Overview

- The RV system
  - Highlights and goals
- Parametric properties
  - What they are, semantics and slicing
- Underlying technologies of RV
  - Monitoring
  - Prediction
  - Mining
- Conclusion

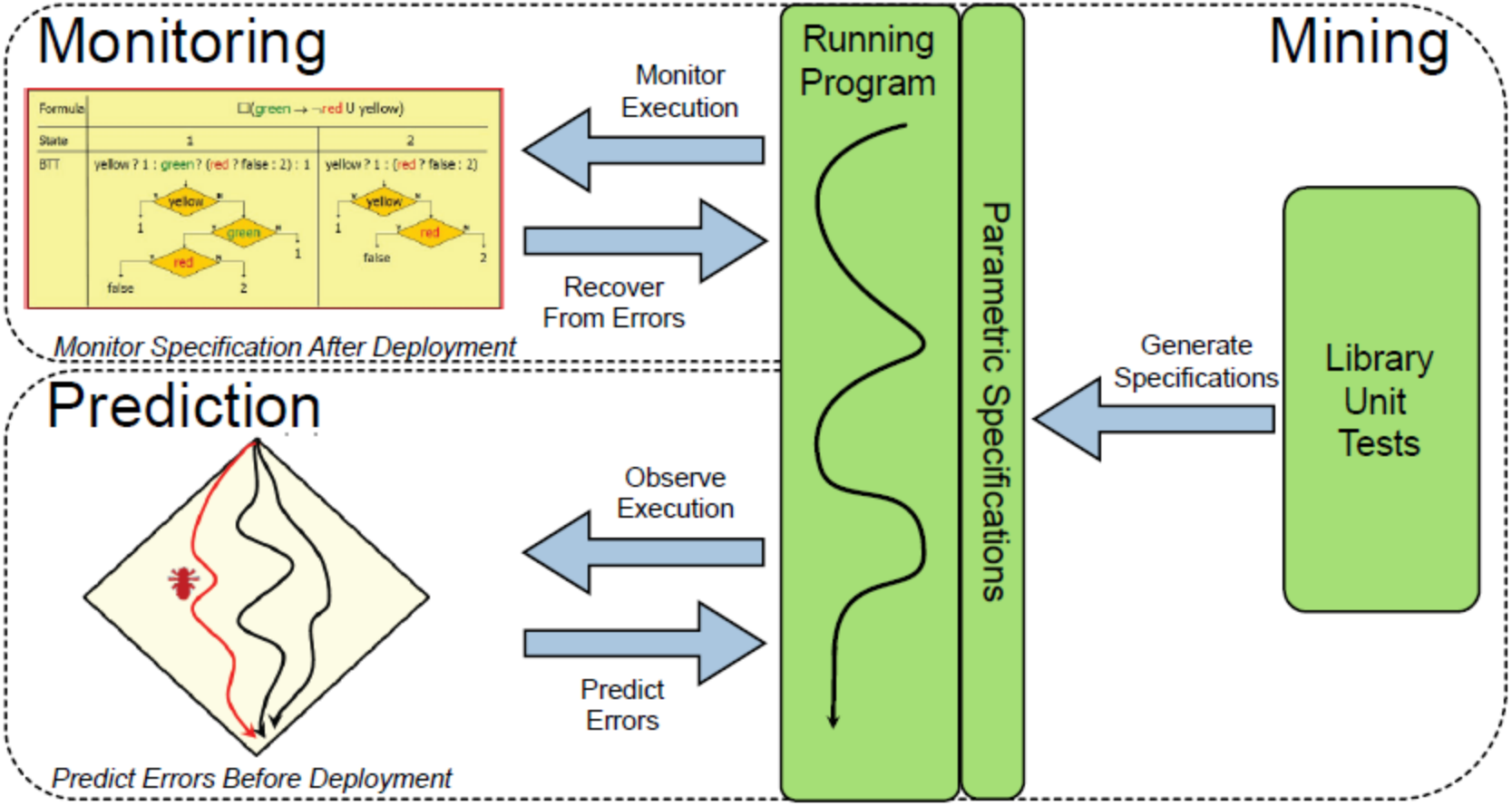
The RV System

- Developed by Runtime Verification, Inc., a startup company in Urbana, in collaboration with the Formal Systems Lab (FSL) at UIUC
- Aims at overcoming the current limitations of runtime verification and, implicitly, at becoming the best runtime verification system
- Builds upon technologies developed during the last 7 years within the FSL@UIUC

How RV System Overcomes Current Limitations of Runtime Verification

- Low runtime and memory overhead: efficient instrumentation; delay monitor creation; garbage collect monitors
- Increase coverage: predictive runtime analysis used to analyze causal models instead of flat execution traces; causal models comprise many traces
- Mine specifications: running and observing unit tests, it learns (1) the most likely interacting events, and (2) the most likely properties they obey

RV System Overview



RV Builds Upon UIUC Technologies

- Efficient monitoring of parametric properties
  - **JavaMOP**
    - RV'03-10, TACAS'05, OOPSLA'07, ASE'08, ASE'09, PLDI'11
- Predictive runtime analysis
  - **jPredictor**
    - FSE'03, TACAS'04, FMOODS'05, CAV'07, SAS'07, ICSE'08
- Mining of parametric properties
  - **jMiner**
    - ICSE'11

RV at Work: Monitoring I

- Consider the following (wrong) Java program:

```
bash-3.2$ more SafeEnum_1.java
import java.util.*;
public class SafeEnum_1 {
        public static void main(String[] args){
                Vector<Integer> v = new Vector<Integer>();
                v.add(1); v.add(2); v.add(4); v.add(8);
                Enumeration e = v.elements();
                int sum = 0;
                if(e.hasMoreElements()){
                        sum += (Integer)e.nextElement();
                        v.add(11);
                }
                while(e.hasMoreElements()){
                        sum += (Integer)e.nextElement();
                }
                v.clear();
                System.out.println("sum: " + sum);
        }
}
```

Wrong!

- It modifies vector while enumerating it

RV at Work: Monitoring II

- Write parametric specifications under ./mop:

```
bash-3.2$ more mop/SafeEnum.mop
package mop;
import java.io.*;
import java.util.*;

SafeEnum(Vector v, Enumeration e) {
    event create after(Vector v) returning(Enumeration e) :
            call(Enumeration Vector+.elements())
         && target(v) {}
    event updatesource after(Vector v) :
            (call(* Vector+.remove*(..))
         || call(* Vector+.add*(..))
         && target(v){}
    event next before(Enumeration e) :
            call(* Enumeration+.nextElement())
         && target(e){}

    ere : create next* updatesource+ next
    @match {
        System.out.println("improper enumeration usage at " + __LOC);
        __RESET;
    }
}
```

parameters

instrumentation

property

reaction

RV at Work: Monitoring III

- Then call RV on the Java program to monitor:

```
bash-3.2$ rv-monitor SafeEnum_1
-Processing ./mop/SafeEnum.mop
 SafeEnumMonitorAspect.aj is generated

SafeEnum_1.java
Executing your program:
improper enumeration usage at SafeEnum_1.java:23
sum: 26
Done
```

- RV compiles the program (with javac), generates monitors as aspects, then weaves them within the program binary (with ajc), then runs the resulting programs

RV at Work: Prediction I

- Consider the following (wrong) Java program:

```
bash-3.2$ more simple/Simple.java
package simple;
public class Simple extends Thread {
   static public int i = 1;
   public static void main(String[] args) {
      (new Simple()).start();
      (new Simple()).start();
   }
   public void run() {
       i++;
       System.out.println(i);
   }
}
```

- Program has a race on the i and a race on output; these races will most likely not appear during testing; RV can predict them from one “successful” execution

RV at Work: Prediction II

- By default, RV prediction searches for races:

```
bash-3.2$ javac simple/Simple.java
bash-3.2$ rv-predict simple.Simple
Instrumenting...
...Done
Executing the instrumented code...
  2
  3
...Done
Running jpredictor...
  Determining race candidates
  The following are our race candidates:
   | java.io.PrintStream (instance #657291792) | simple.Simple.i
  Predicting for race candidate: java.io.PrintStream (instance #657291792)

  /--- Race found on java.io.PrintStream (instance #657291792) ---\
  |  Write at simple.Simple:10                                    |
  |  Write at simple.Simple:10                                    |
  \---------------------------------------------------------------/

  Predicting for race candidate: simple.Simple.i

  /--- Race found on simple.Simple.i ---\
  |  Read  at simple.Simple:10          |
  |  Write at simple.Simple:9           |
  \-------------------------------------/

...Done
bash-3.2$
```

RV at Work: Prediction III

- To predict violations of parametric properties, one needs to place them under ./mop, same like when monitoring them
- A program may have races but those may not lead to violations of specifications
  - Though not a good idea to allow races anyway
- Also, a program may run "successfully", have no races predicted in it, but still violate behavioral specifications

RV at Work: Mining

- The RV miner is less obvious to use
- It works in two stages:
  - First, it observes executions of (small and tricky) unit tests to learn groups of parameters and corresponding events that interact
  - Then it observes (large and likely correct) programs making use of the learned events and
    - It slices the observed (large) parametric traces into (typically lots of small) non-parametric traces
    - It learns non-parametric properties that best explain the computed slices
    - Then adds the learned parameters to them

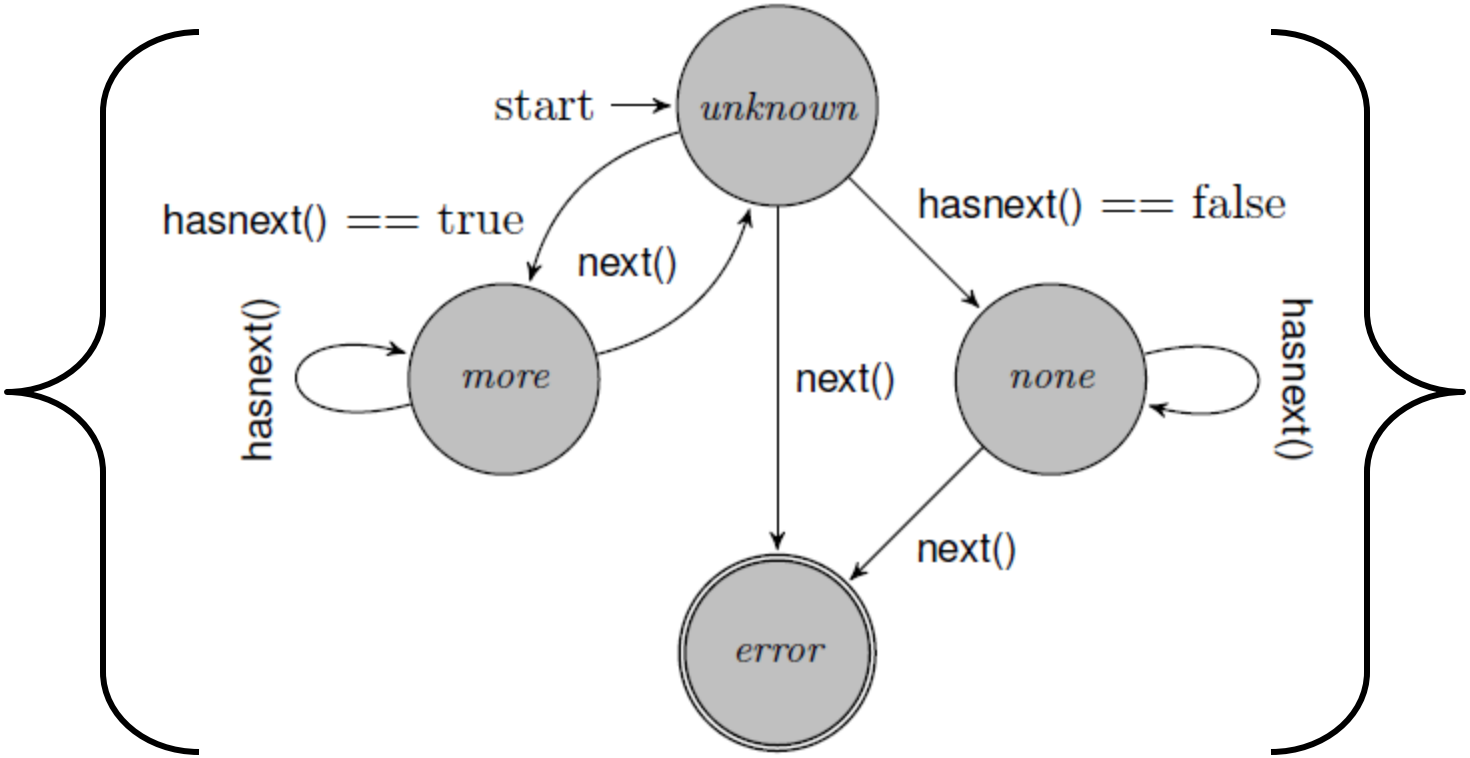
Parametric Properties

- At the core of RV: all requirements are specified as parametric properties
- These are properties over parametric events
- Comprise a potentially unbounded number of property instances, one for each parameter
- The main challenge is how to efficiently and correctly keep track of all property instances

Examples of Parametric Properties I

- Typestates are particular, one-parameter properties
- For example, “hasnext() must be called and hold before each next() is called”, can be defined as follows:

$$\Lambda i\ .$$



Examples of Parametric Properties II

- Same "hasnext() must be called and hold before each next() is called" typestate property, but specified using different parametric variants:
  - As a parametric regular expression:

$$\Lambda i\,.\,(\mathsf{hasnexttrue}\langle i\rangle^{+}\mathsf{next}\langle i\rangle \mid \mathsf{hasnextfalse}\langle i\rangle^{*})^{*}$$

  - As a parametric linear temporal logic formula:

$$\Lambda i\,.\,\Box(\mathsf{next}\langle i\rangle \implies \odot\mathsf{hasnexttrue}\langle i\rangle)$$

Examples of Parametric Properties III

- Colections are not allowed to change while accessed through iterators:

$$\Lambda c, i\,.\ \ \mathsf{createIter}\langle c\, i\rangle\ \ \mathsf{next}\langle i\rangle^*\ \ \mathsf{updateColl}\langle c\rangle^+\ \ \mathsf{next}\langle i\rangle$$

- To avoid clutter, we drop the understood event parameters, i.e., we write the above as

$$\Lambda c, i\,.\ \ \mathsf{createIter}\ \ \mathsf{next}^*\ \ \mathsf{updateColl}^+\ \ \mathsf{next}$$

Examples of Parametric Properties IV

- Authenticate each key before use:

$$\Lambda k \,.\, \Box(\mathsf{use} \longrightarrow \diamond\mathsf{authenticate})$$

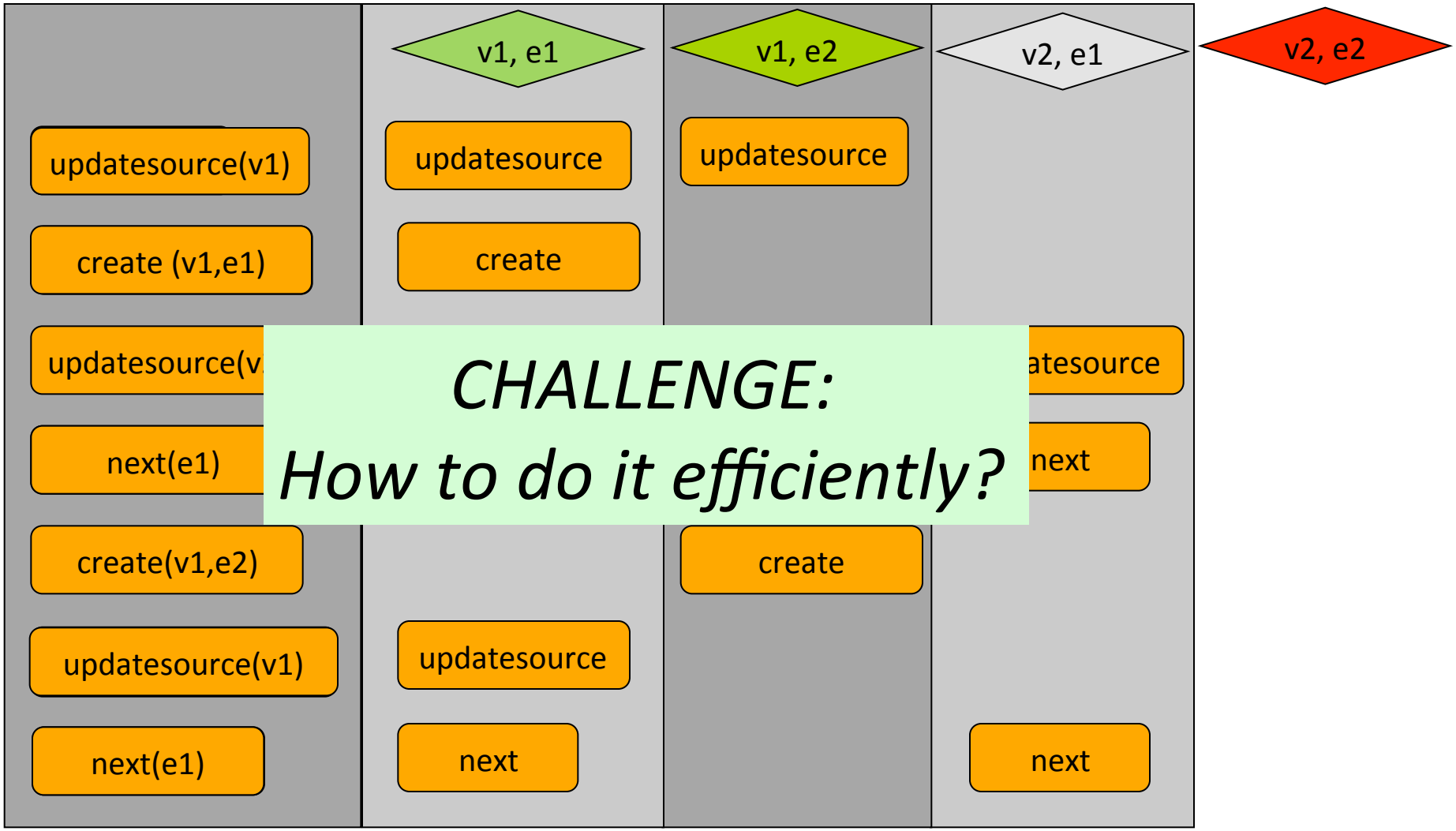
- Each function should release each lock as many times as it acquires it (this is a parametric context-free property):

$$\Lambda l \,.\, S \longrightarrow S\ \mathsf{begin}\ S\ \mathsf{end} \mid S\ \mathsf{acquire}\ S\ \mathsf{release} \mid \epsilon$$

Parametric Trace Slicing I

- RV is all about monitoring, predicting violations of, and mining parametric properties
- All the above work with parametric traces
  - Traces formed with parametric events
- Therefore, all face the problem of trace slicing
  - How to identify the sub-traces of events corresponding to each parameter instance?

Parametric Trace Slicing II



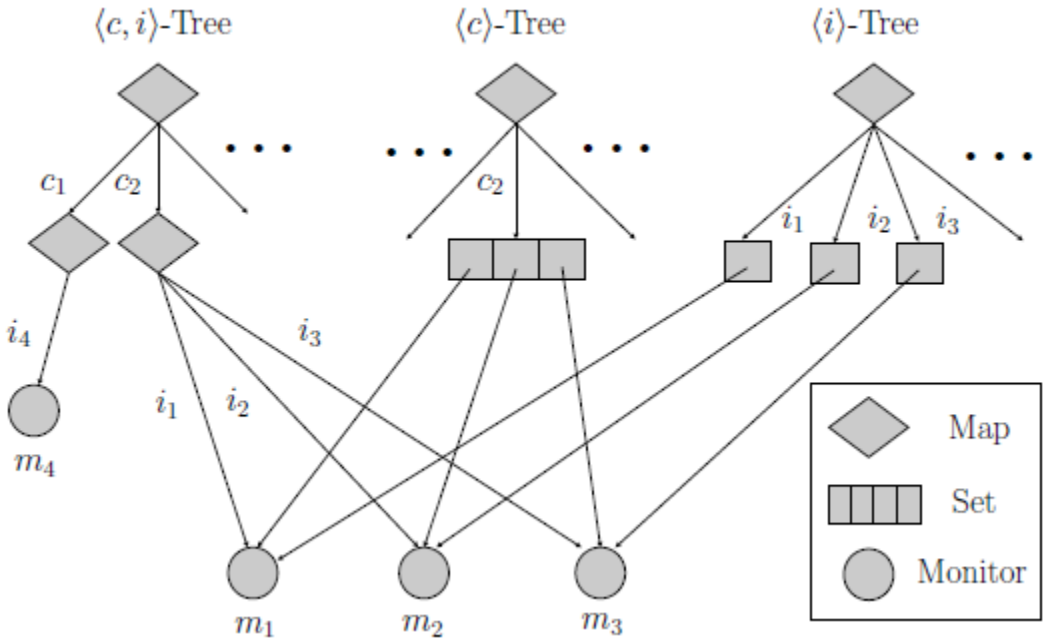
For given parameters (v, e)

RV Monitoring

- Sends each trace slice to one monitor instance
- Problem: The number of slices can be huge (potentially unbounded)
- Challenge: Manage monitors efficiently
  - Index them for efficient retrieval
  - Garbage collect them when become unnecessary

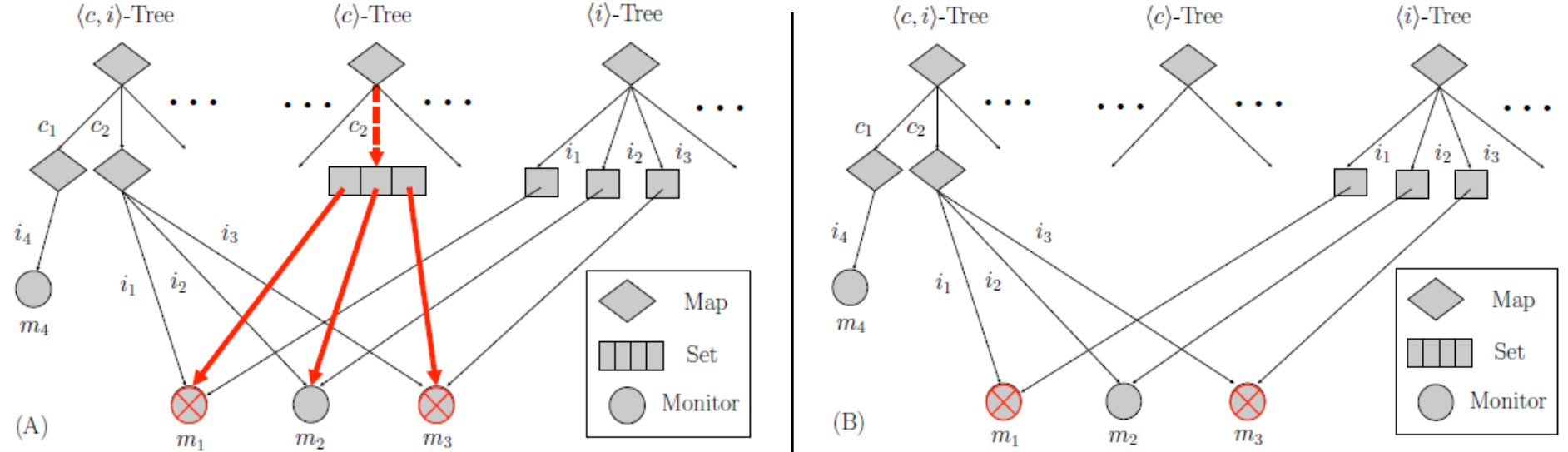
Monitor Indexing

- Indexing trees using weak references
  - One tree per parameter combination in events
  - Monitors are collected when no events available



More Garbage Collection

- Indexing trees alone not good enough
- Pathological examples where unnecessary monitors stay alive forever
- Solution: analyze the property statically; then collect monitors when they have no future



RV Monitoring Performance Evaluation

>2x faster than JavaMOP, >10x faster than Tracematches

(A) Runtime overhead in % (B) Memory overhead in MB

	ORIG (sec)	HASNEXT			UNSAFEITER			UNSAFEMAPITER			UNSAFESYNCCOLL			UNSAFESYNCMAP			ALL
(A)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	3.6	2119	448	116	19194	569	251	∞	1203	178	1359	746	212	1942	716	130	982
jython	8.9	13	0	0	11	0	1	150	18	3	11	1	1	10	0	0	4
avrora	13.6	45	54	55	637	311	118	∞	113	42	75	144	80	54	74	16	275
batik	3.5	3	2	3	355	9	8	∞	8	5	208	9	9	5	3	0	28
eclipse	79.0	-2	4	-1	0	-1	-1	5	-3	0	-4	2	1	∞	-1	-1	0
fop	2.0	200	49	48	350	21	13	∞	58	14	∞	78	25	∞	71	19	133
h2	18.7	89	17	13	128	9	4	1350	21	6	868	21	4	83	20	5	23
luindex	2.9	0	0	1	0	0	1	1	4	1	1	1	1	2	0	0	1
lusearch	25.3	-1	1	0	1	2	2	2	2	0	4	0	1	3	1	1	3
pmd	8.3	176	84	59	1423	162	123	∞	571	188	1818	192	76	∞	144	26	620
sunflow	32.7	47	5	3	7	2	0	9	4	1	13	6	5	17	6	6	6
tomcat	13.8	8	1	1	37	1	1	3	1	1	2	0	1	2	1	3	1
tradebeans	45.5	0	-1	1	1	1	2	5	3	-1	-1	1	2	3	1	5	2
tradesoap	94.4	1	3	0	2	1	1	2	0	1	0	0	1	2	2	5	1
xalan	20.3	4	2	2	27	7	2	10	5	2	3	2	3	4	4	3	4
(B)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	4.9	56.8	19.3	13.2	7.7	146.8	79.0	∞	173.4	56.1	6.8	127.9	48.3	6.9	55.4	12.7	340.9
jython	5.3	5.7	4.6	4.8	4.9	4.6	4.8	6.0	19.5	4.7	5.3	4.5	4.4	5.9	4.8	5.1	4.7
avrora	4.7	4.6	12.4	9.1	4.4	136.2	15.8	∞	14.7	8.5	4.3	28.0	12.6	4.4	13.0	4.9	22.3
batik	79.1	79.2	78.7	79.3	75.2	93.6	86.6	∞	91.2	79.6	78.2	93.2	85.1	79.9	86.9	76.7	104.3
eclipse	95.9	100.8	107.6	97.1	98.3	100.0	110.3	106.9	93.8	101.1	100.4	109.2	90.1	∞	98.6	98.7	98.9
fop	20.7	97.4	47.1	52.5	24.3	24.2	29.4	∞	69.2	28.1	∞	54.8	24.8	∞	55.9	25.2	47.5
h2	265.0	267.8	598.5	565.2	267.2	266.2	262.4	312.4	688.3	268.2	271.4	690.3	265.5	271.0	718.3	270.0	283.7
luindex	6.8	5.6	5.5	5.6	6.3	6.9	6.8	7.4	8.2	6.9	7.4	7.4	7.5	7.1	7.4	11.0	11.8
lusearch	4.6	4.7	4.4	4.8	4.6	4.8	4.2	4.0	4.3	4.8	4.5	4.5	4.6	4.6	4.8	4.7	4.7
pmd	18.0	56.9	59.8	48.5	17.2	146.3	86.4	∞	212.7	93.6	20.3	238.4	84.6	∞	117.1	32.9	420.0
sunflow	4.4	4.5	4.8	4.9	4.8	4.3	4.7	4.7	4.4	4.4	5.1	4.3	4.9	4.5	4.7	4.5	4.6
tomcat	11.6	11.4	12.3	11.4	12.5	11.0	11.5	11.9	11.4	11.0	11.3	11.3	11.3	11.4	11.4	11.8	11.8
tradebeans	63.2	62.9	62.7	62.1	63.7	63.9	64.1	63.3	62.5	62.7	63.2	62.8	62.0	64.0	62.8	64.0	62.5
tradesoap	64.1	61.8	62.3	63.3	63.4	63.1	64.4	64.1	63.5	62.0	60.7	65.0	65.9	65.5	64.5	65.6	64.5
xalan	4.9	4.9	5.0	5.1	4.9	4.9	4.9	4.9	4.5	4.9	5.0	4.8	5.0	5.1	4.9	4.9	5.0

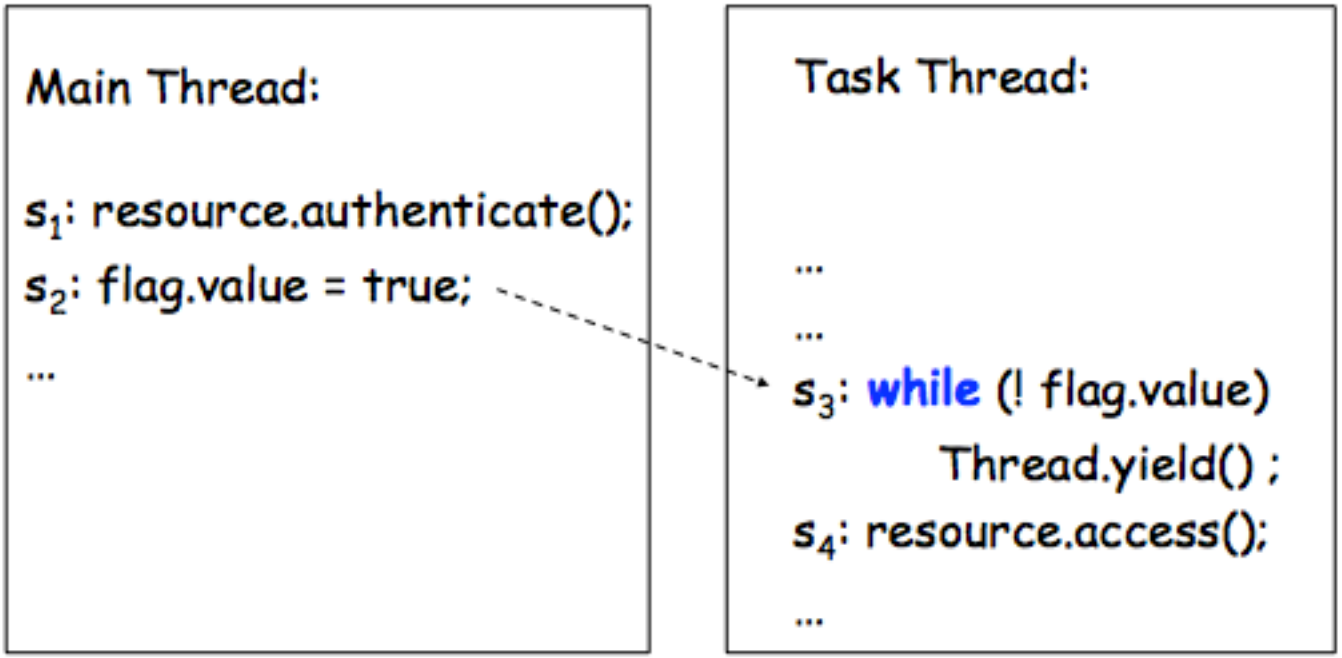
RV Prediction

- Predicts errors in concurrent programs
  - Model checking: number of interleavings is prohibitively large
  - Testing: interleavings depend on environment
- Combine dynamic and static methods to find bad behaviors near correct executions

29

Prediction Example

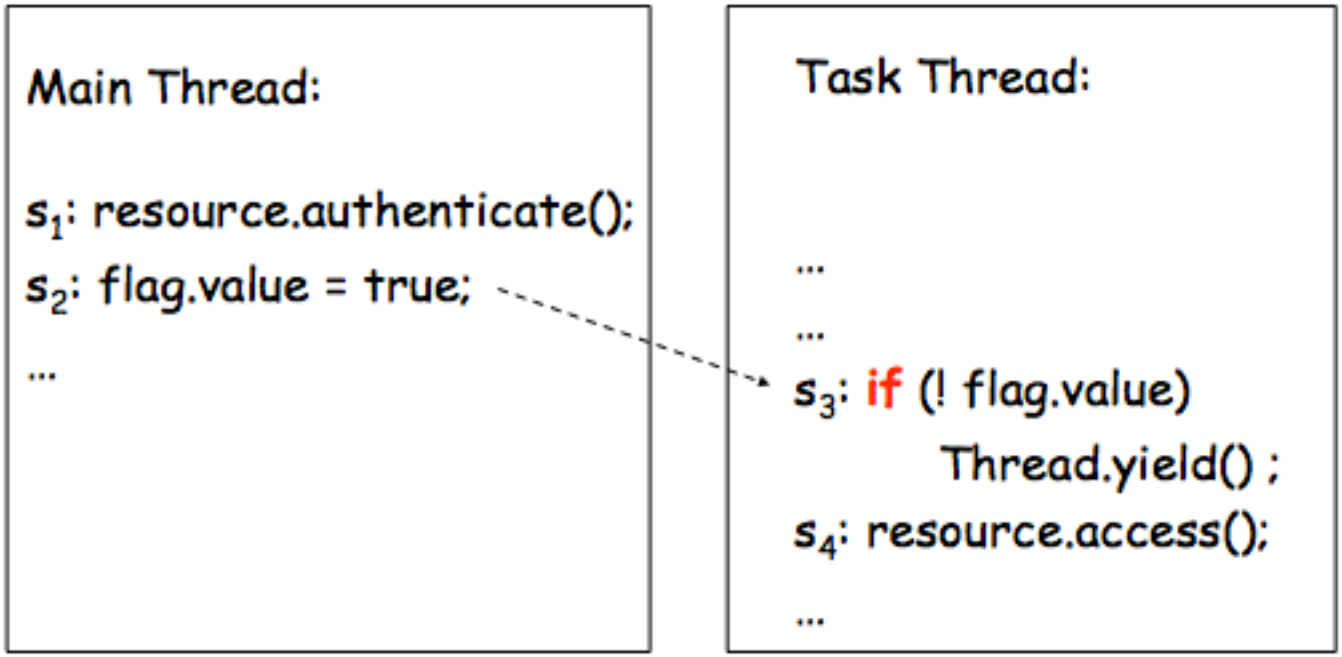
Property: "authenticate before access"



Observed execution: ... s_1 s_2 s_3 s_4 ...

Prediction Example

Property: "authenticate before access"

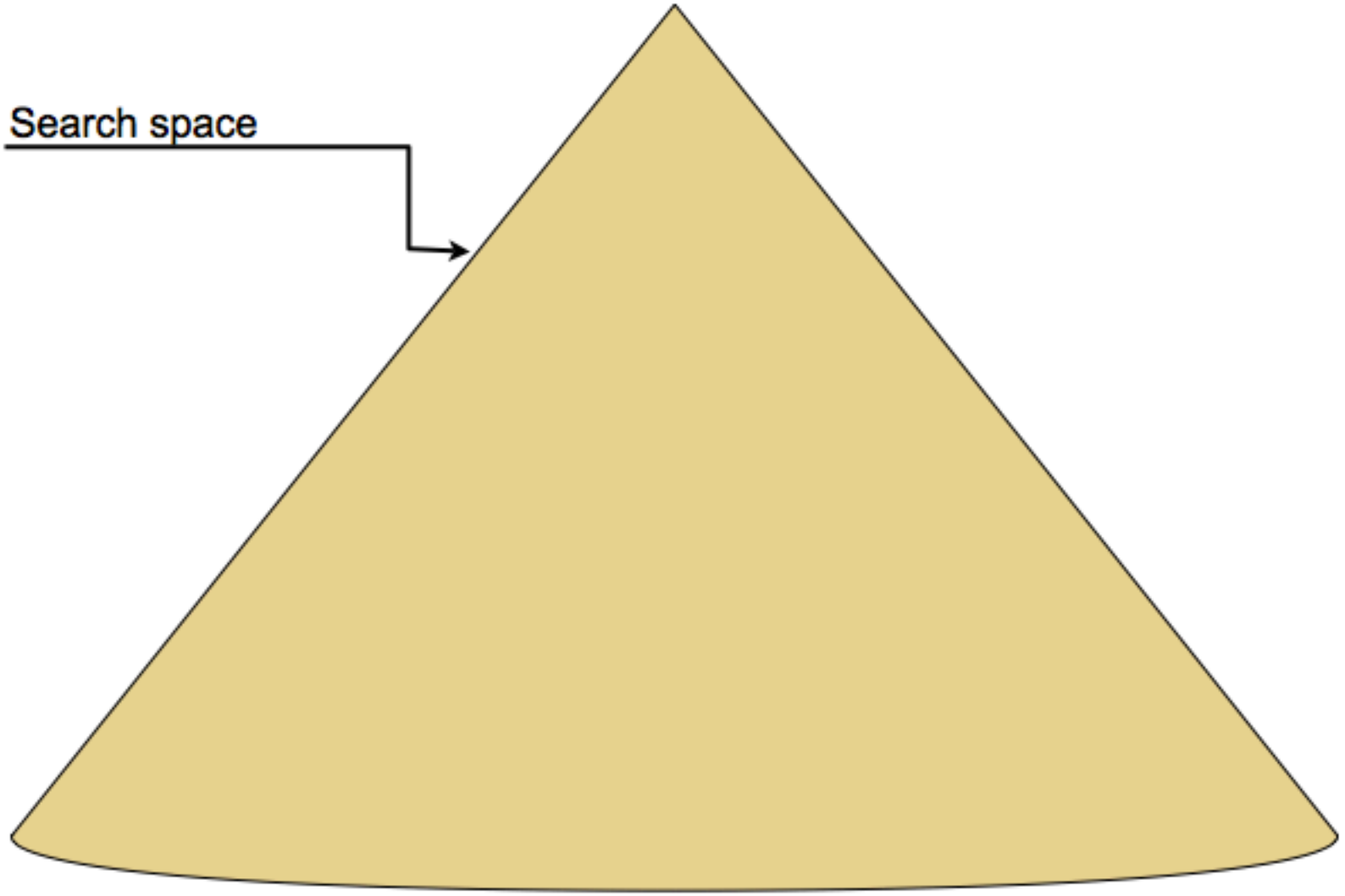


Observed execution: ... s_1 s_2 s_3 s_4 ...

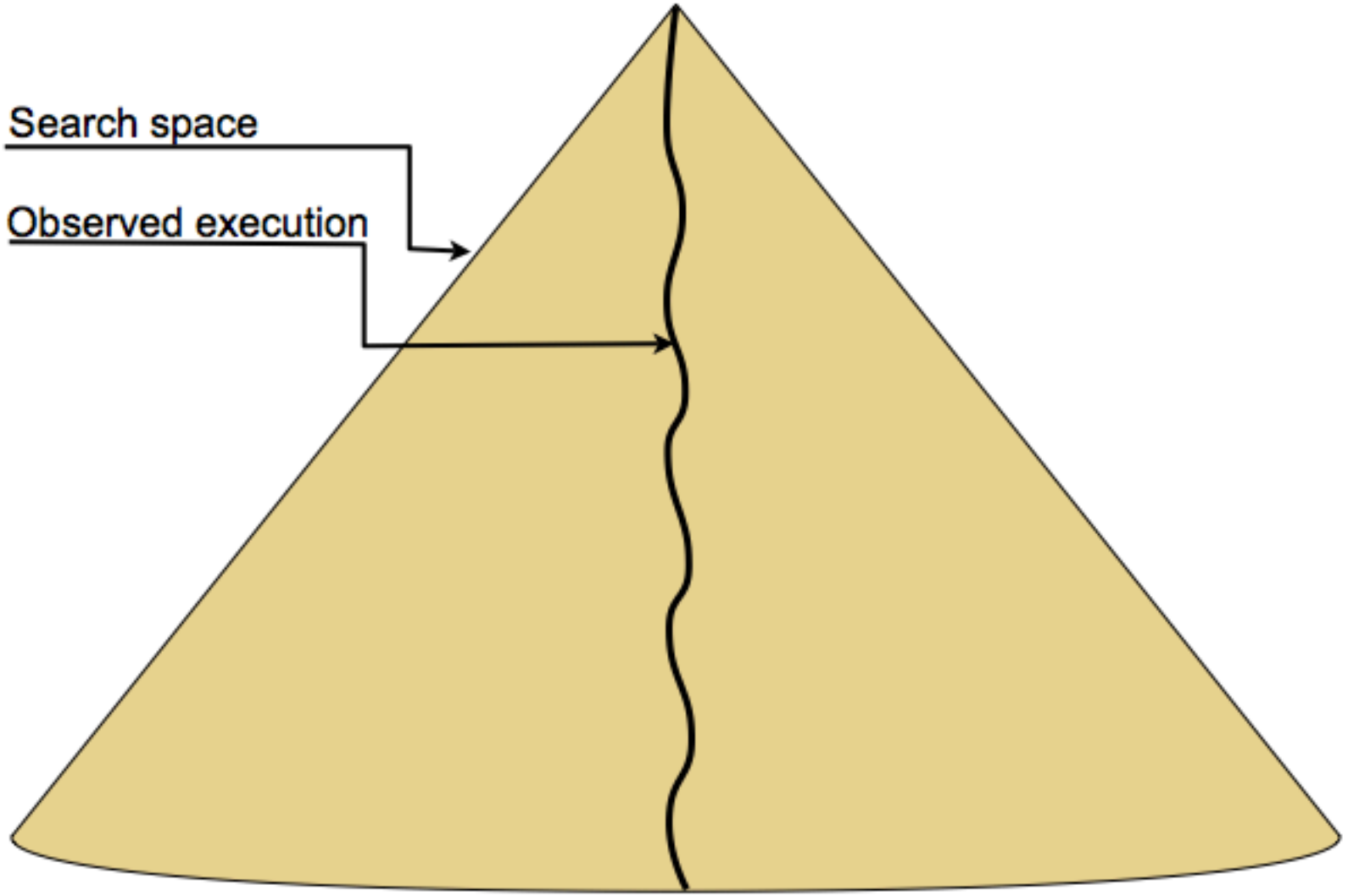
- Buggy S_4 can be executed before S_1
- Low likelyhood to hit error in testing

31

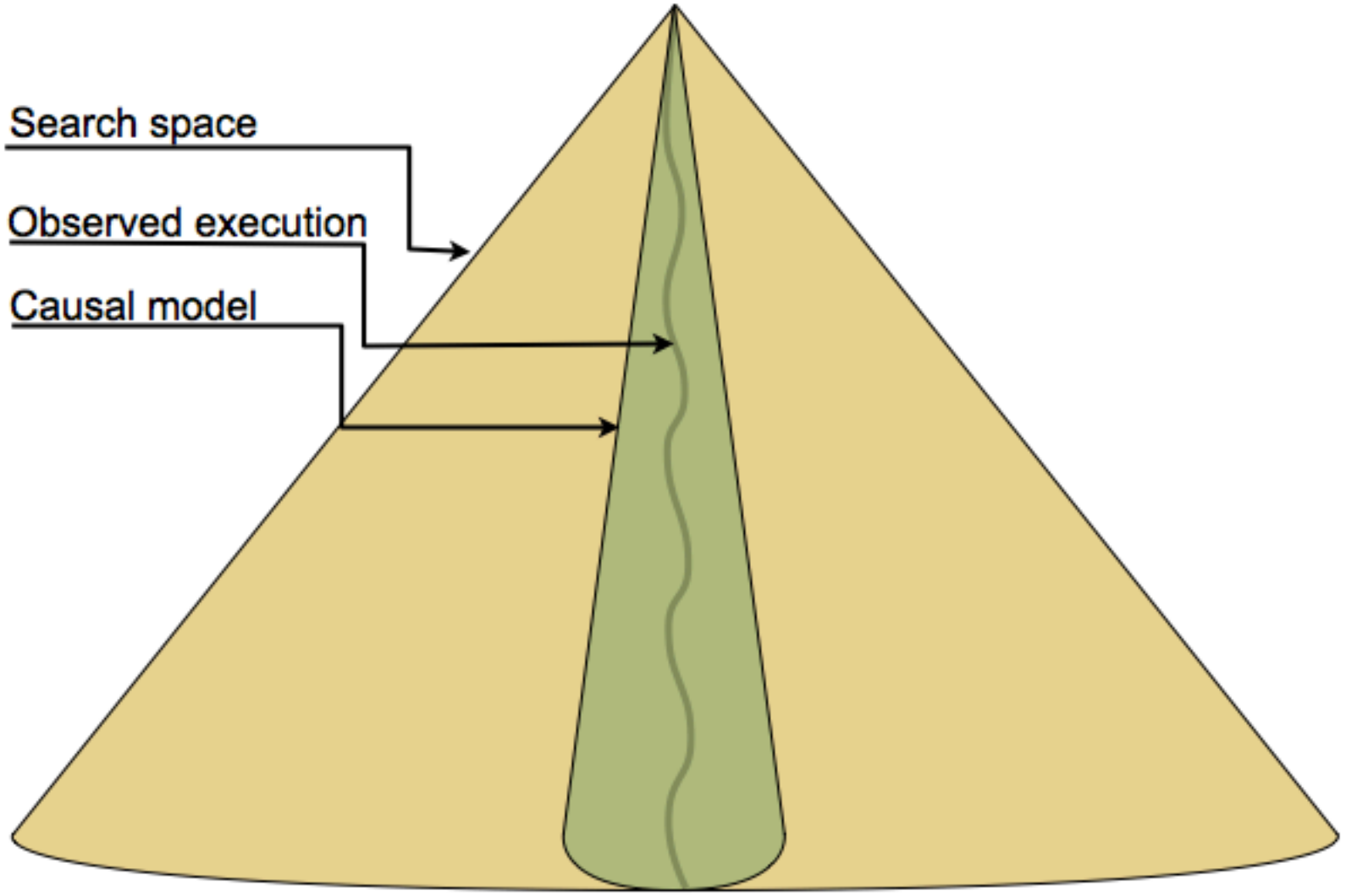
Predictive Runtime Analysis



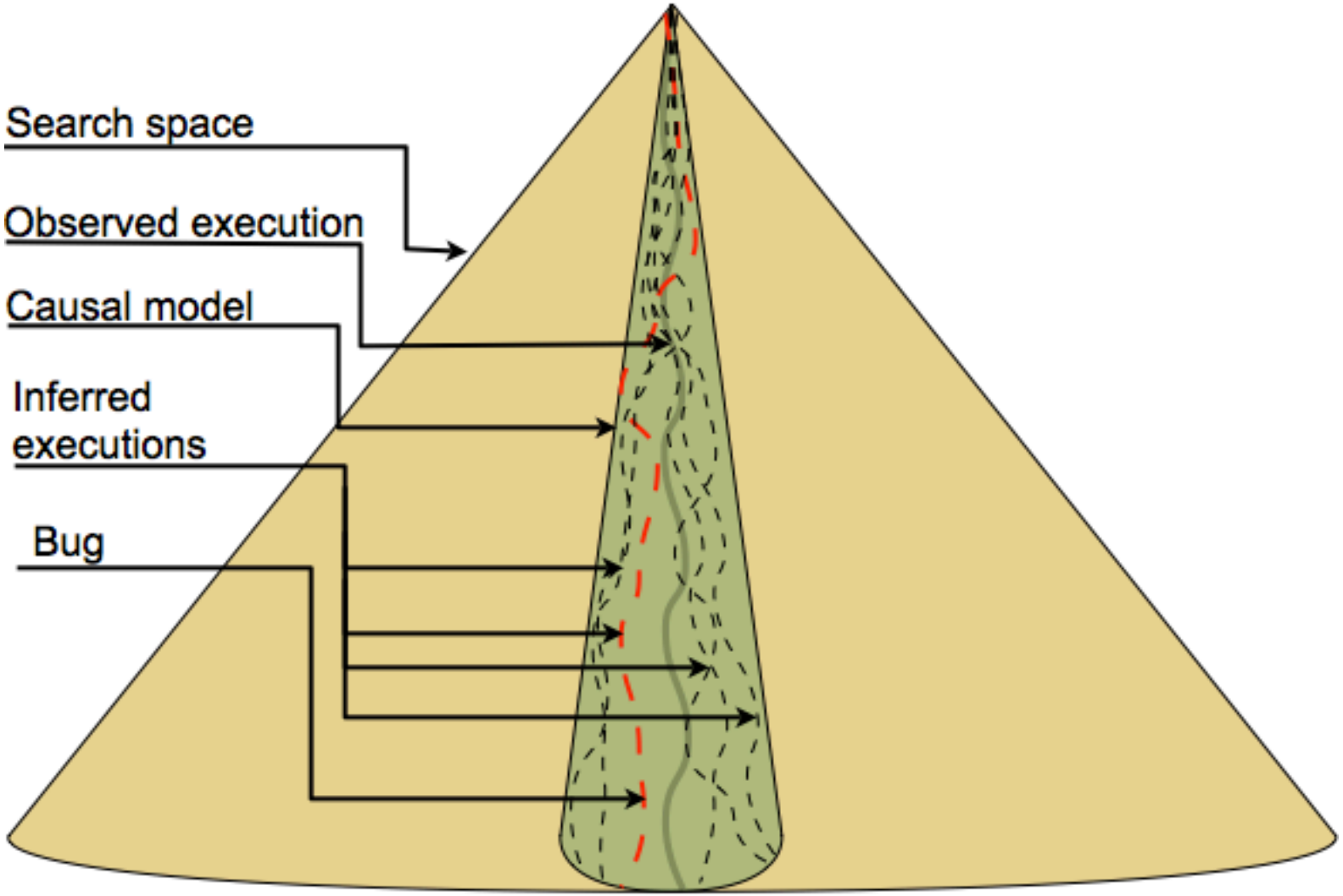
Predictive Runtime Analysis



Predictive Runtime Analysis

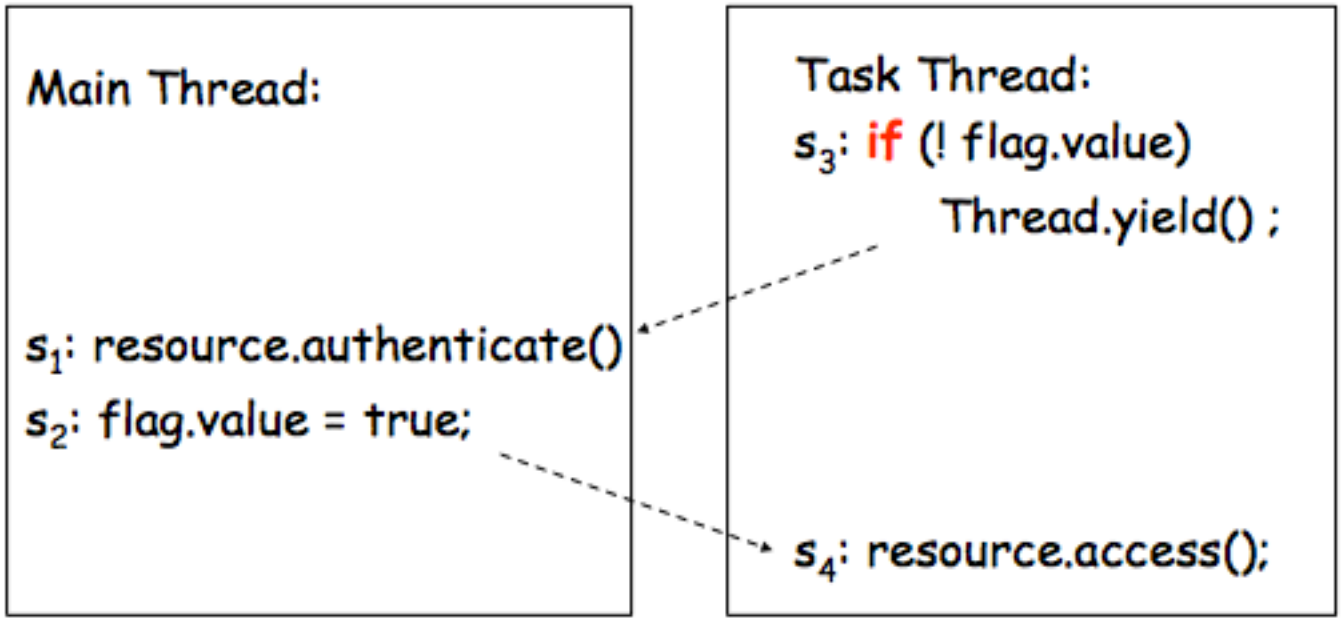


Predictive Runtime Analysis



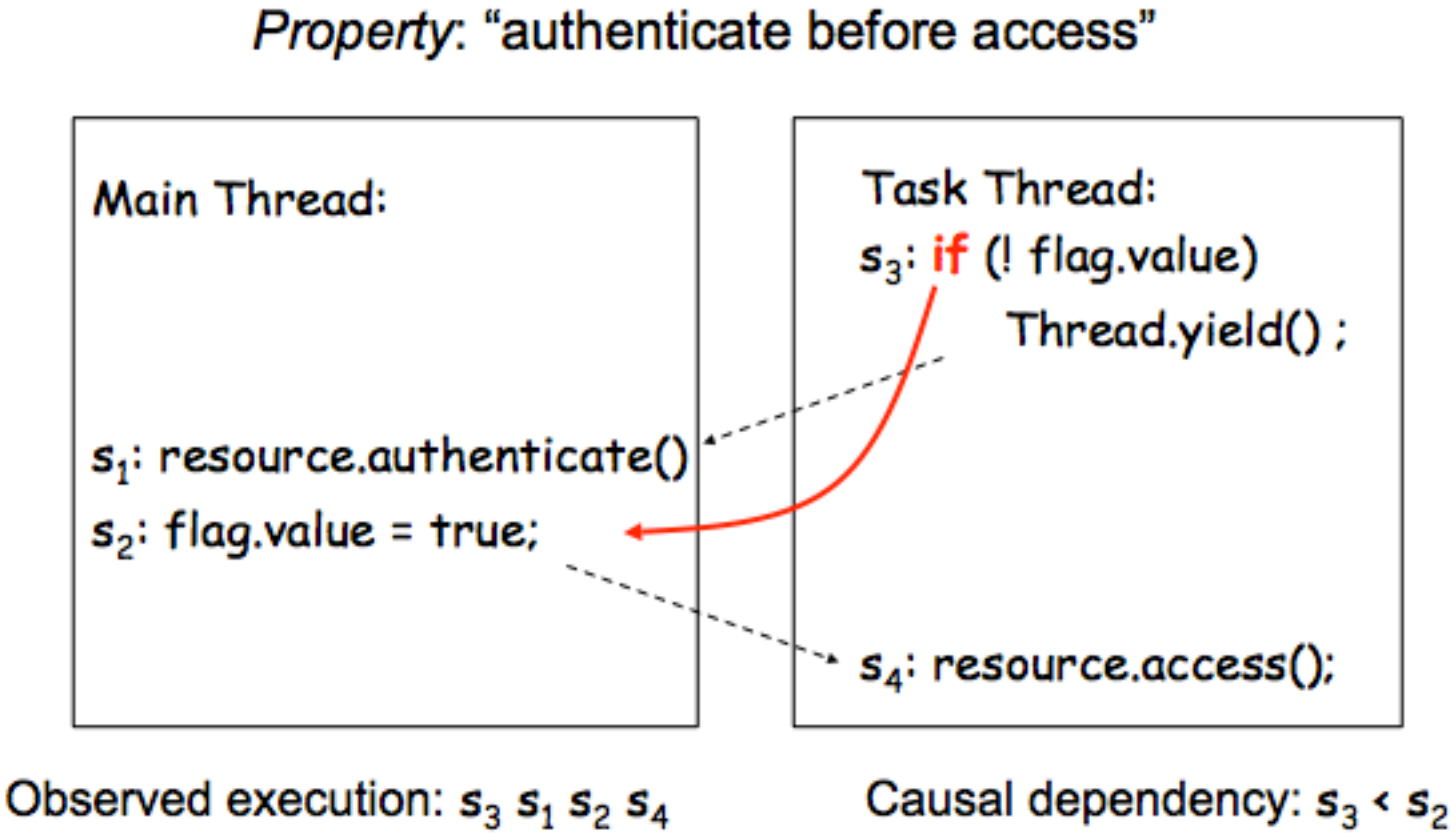
Happens-Before Works ... If Lucky

Property: "authenticate before access"



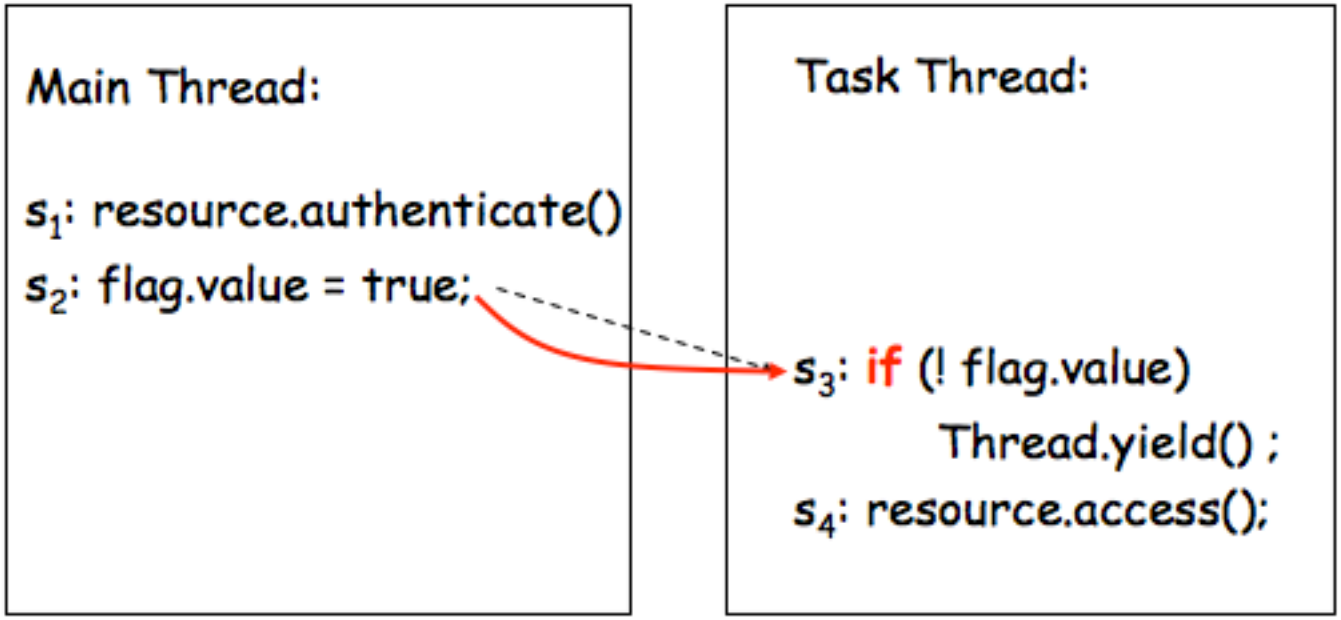
Observed execution: s_3 s_1 s_2 s_4

Happens-Before Works ... If Lucky



Happens-Before Limitations

Property: "authenticate before access"



Observed execution: $s_1\ s_2\ s_3\ s_4$

Causal dependency: $s_2 < s_3$. No bug found …

Too constrained: access will be performed regardless of the flag

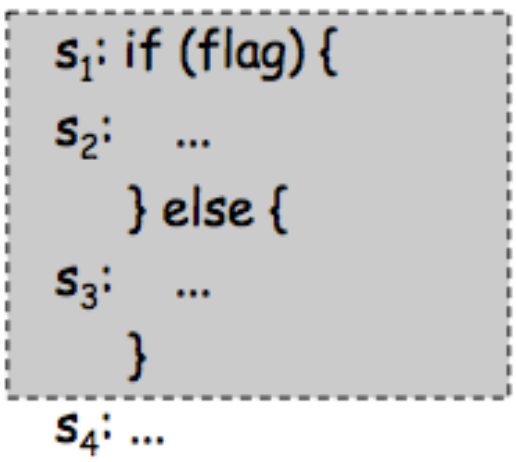
RV Uses Sliced Causality

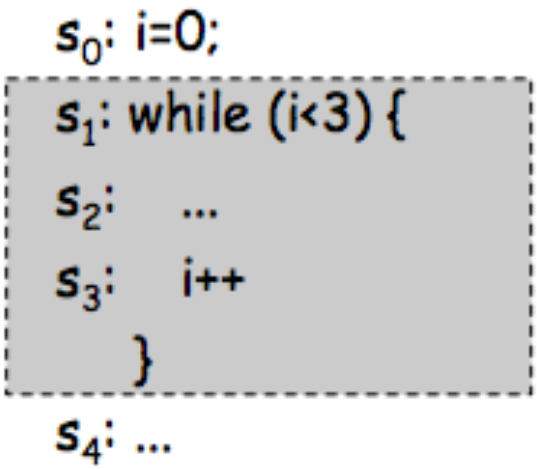
- Relaxes the Happens-Before causal model
- How? Focus on the property
- Use static information about the program
- Remove irrelevant events and causalities
  - Smaller and more relaxed causal model
  - (Exponentially) more inferred executions
  - Better predictive capability

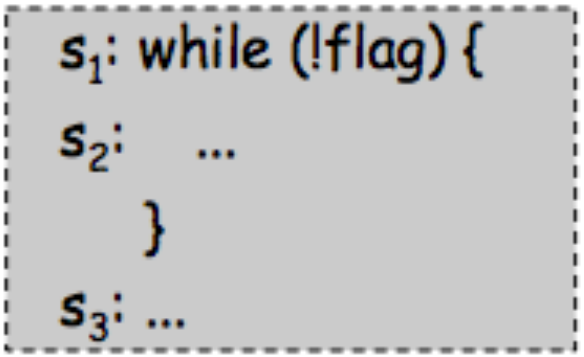
39

Static Information: Control Scope

- S_2 is in the control scope of S_1 if its execution depends on a choice at S_1





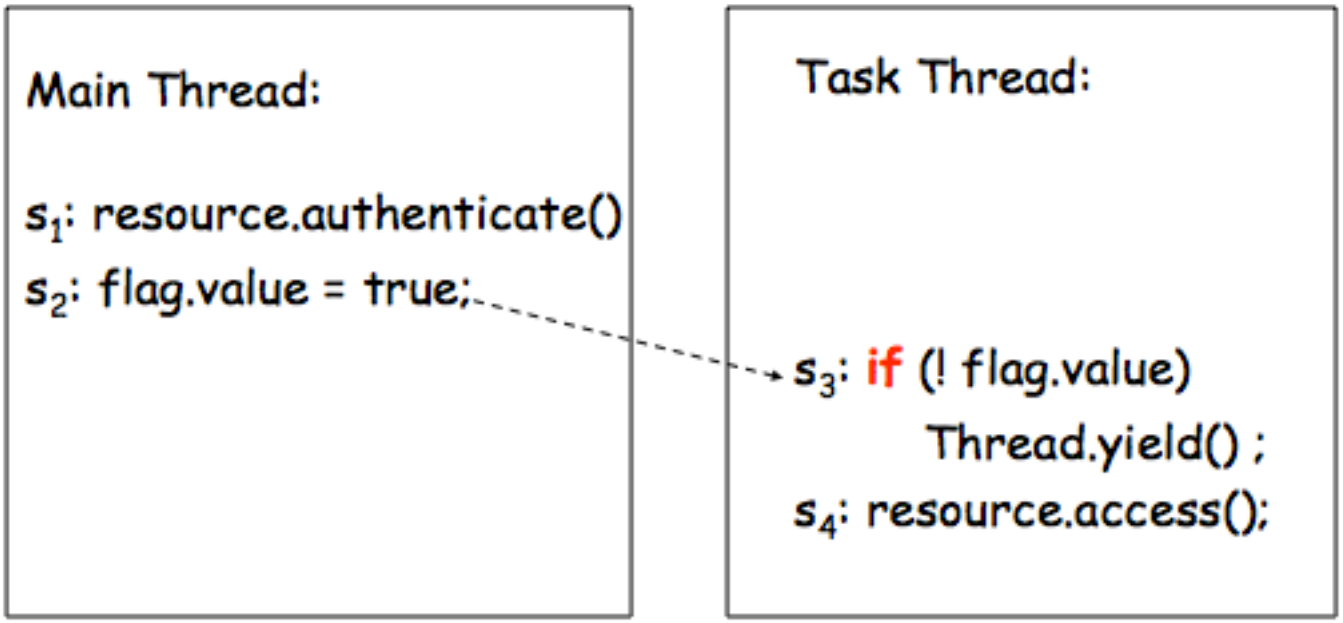


- Extends to other control statements
  - break/continue, return, exceptions

40

Sliced Causality Works!

Property: "authenticate before access"

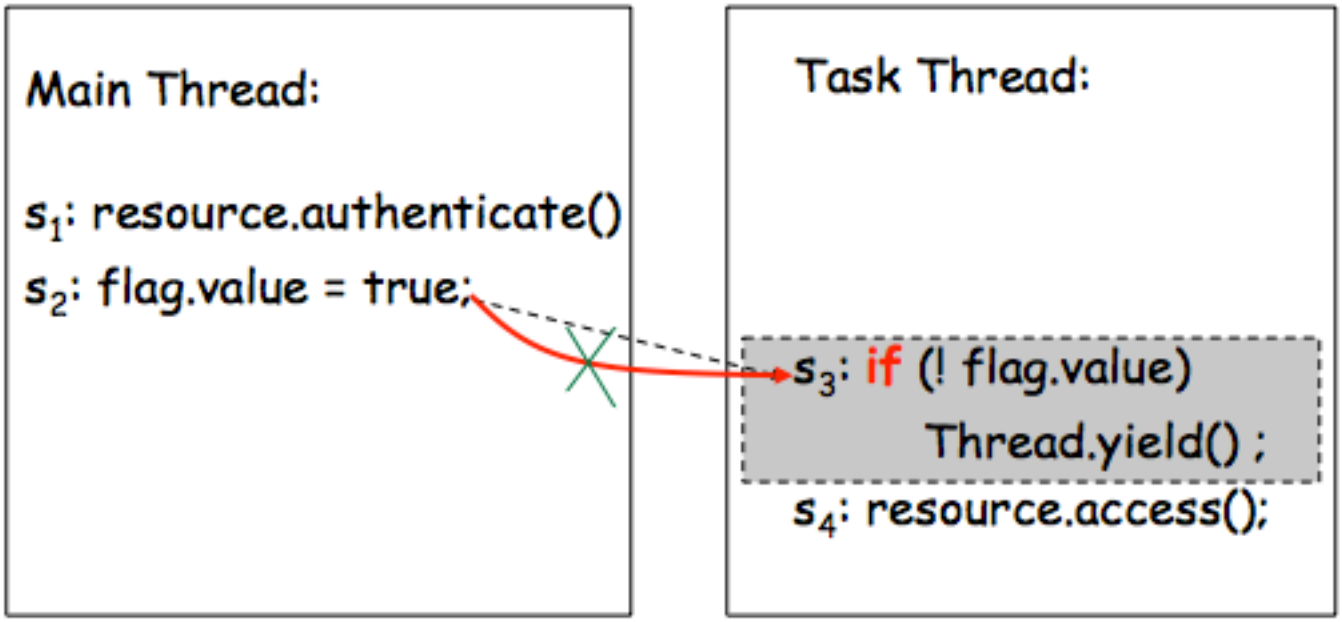


Observed execution: s_1 s_2 s_3 s_4

Only s_1 and s_4 directly relevant to the property

Slice Causality Works!

Property: "authenticate before access"



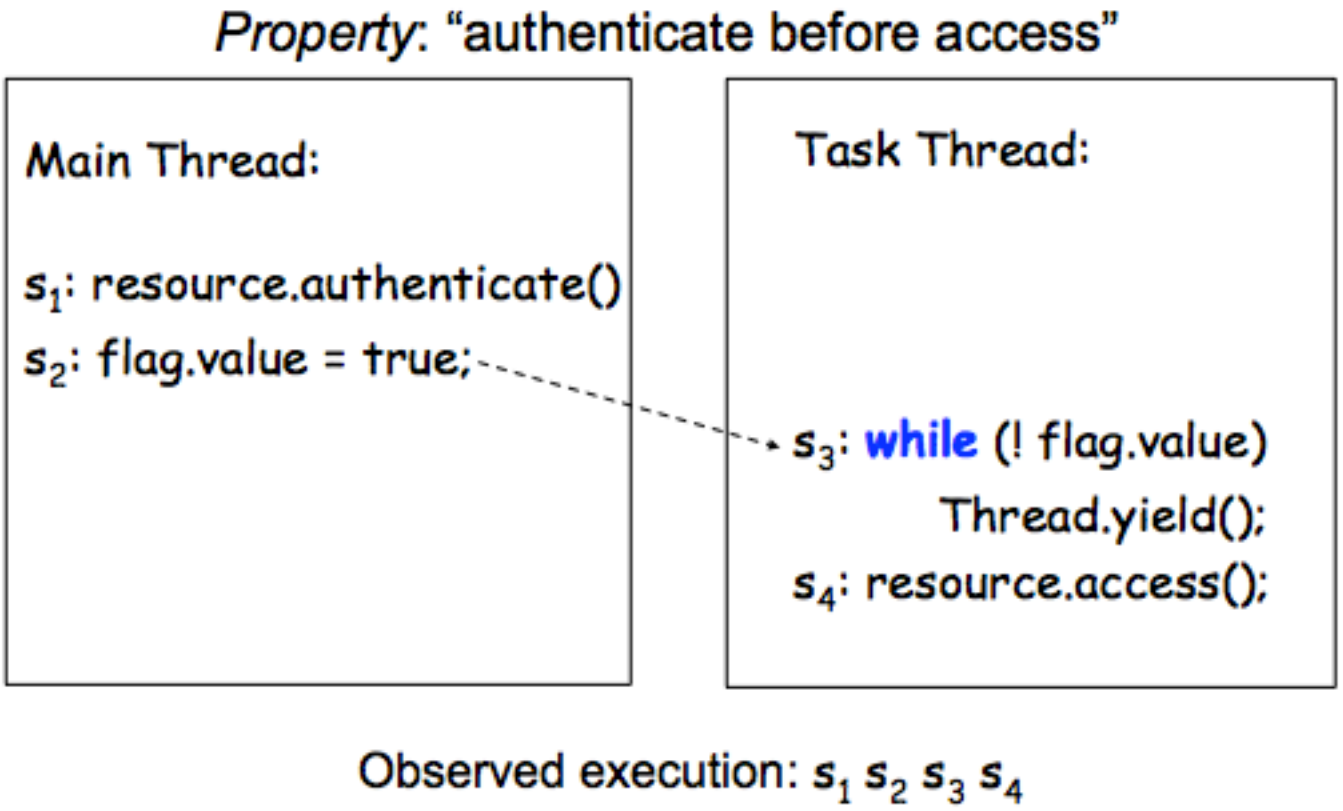
Observed execution: $s_1\ s_2\ s_3\ s_4$

Only s_1 and s_4 directly relevant to the property

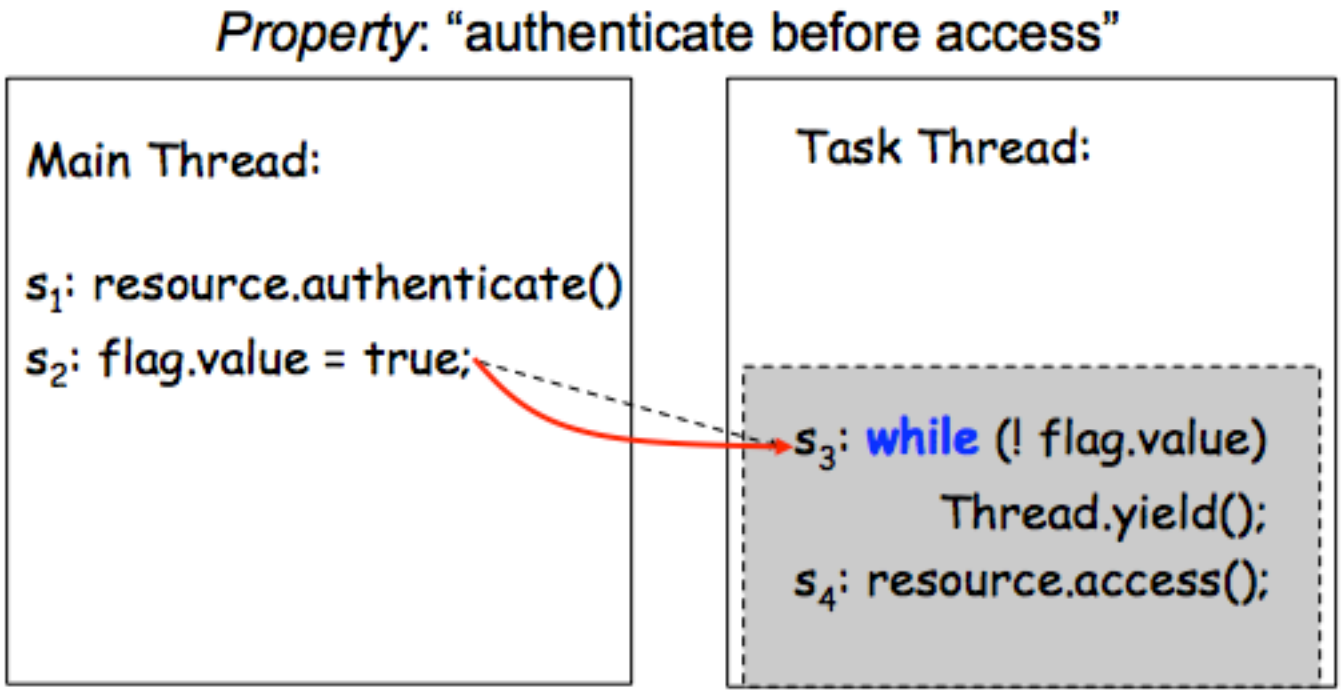
Execution of s_4 not dependent of s_3; ignore the causal dependency $s_2 < s_3$

Sliced causality: $s_1 <> s_4$; $s_4\ s_1$ is a potential execution. Bug detected!

No False Alarms ☺



No False Alarms ☺



Observed execution: $s_1\ s_2\ s_3\ s_4$

Execution of s_4 depends on flag.value being *true* at s_3

causal dependency $s_2 < s_3$ matters

Sliced causality: $s_1 < s_2 < s_3 < s_4$, no false alarm!

RV Prediction Performance

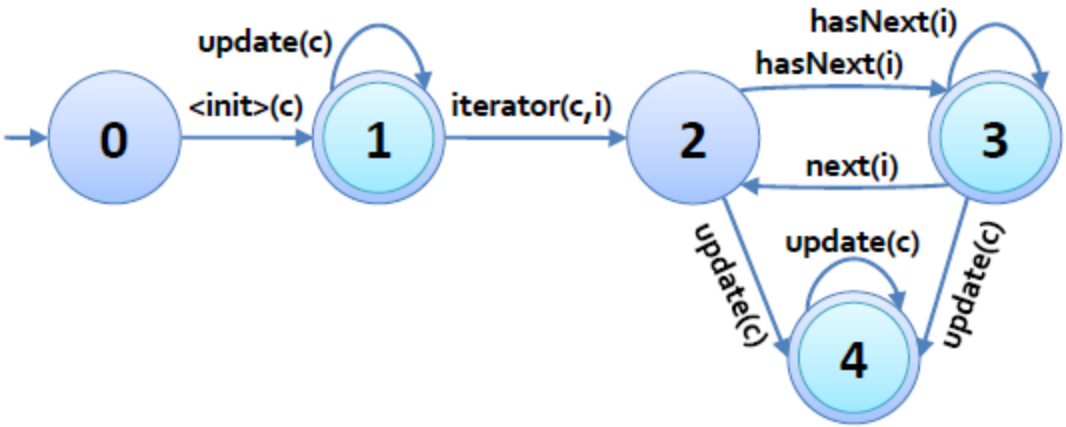
		jPredictor		rv-predict	
Name	Input	Real Time	Disk Usage	Real Time	Disk Usage
account	-	0:02.57	236K	0:06.07	364K
elevator	-	5:55.29	63M	1:20.31	864K
tsp	map4 2	4:24.44	16M	1:45.22	744K
tsp	map5 2	8:12.31	17M	2:45.28	868K
tsp	map10 2	> 3 hours	> 230M	33:45.32	2.8M
huge	-	crash	crash	0:42.22	13M
medium	-	crash	crash	0:06.12	840K
small	-	crash	crash	0:05.99	292K
mixedlockshuge	-	> 2 hours	> 250M	0:05.68	2.9M
mixedlocksbig	-	4:39.08	25M	0:05.68	496K
mixedlocksmedium	-	0:08.92	2.7M	0:07.25	308K
mixedlockssmall	-	0:05.46	1.5M	0:05.67	296K

RV Mining

- RV monitoring slices parametric traces and sends each slice to a monitor instance
- RV miner is dual to RV monitor:
  - It uses (almost) the same trace slicer, but instead of sending the slices to monitors, uses them as input to a regular property learner (PFSA)
- This way, we were able to mine most of the parametric specifications that we previously used for monitoring

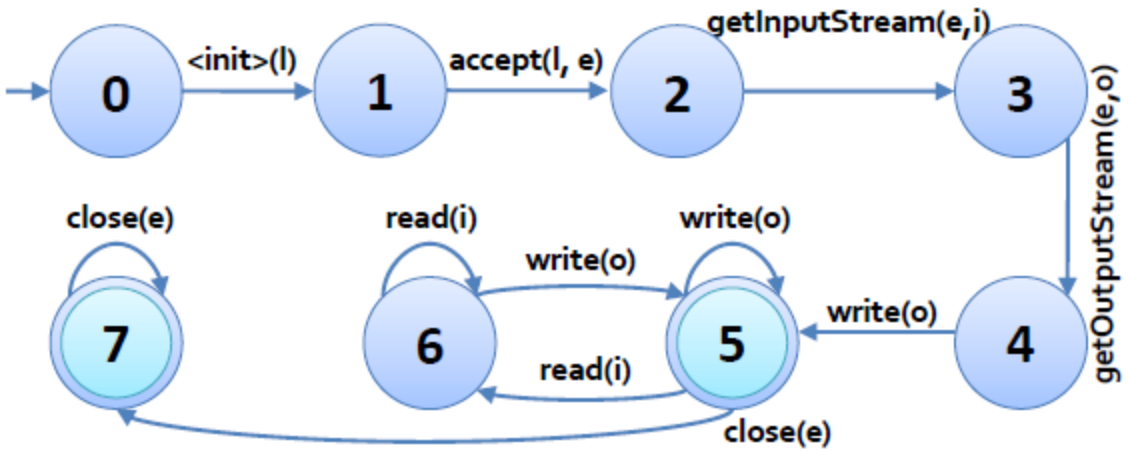
RV Mining Example I

Two properties in one:
- the collection-iterator
- the hasnext typestate



RV Mining Example

- Server socket parametric specification:



RV Mining Preliminary Evaluation I

- Used several packages that come with unit tests

Target package	Event Specifications		Param. Specifications	
	# files	# events	# files	# events
java.util	370	65,854,349	14	88,999,435
java.io	382	28,835,588		
java.lang	372	41,784,568		
java.net	221	9,429,744	31	10,938,168

(a) Traces used in the experiments

Target package	# event specifications	# parametric specifications
java.io	145	66
java.lang	82	48
java.net	90	36
java.util	181	80

(b) Inferred events and mined specifications

RV Mining Preliminary Evaluation II

- Still relatively slow, though the mined specifications are very useful and general-purpose
- Slicing is the most expensive

Target package	Event Specifications	Param. Specifications	
		Slicing	Learning
java.io	24	115	24
java.lang	38	112	75
java.net	59	14	1
java.util	59	133	86

(c) Times (minutes; Windows, 3GHz, 1GB)

Conclusion

- The RV system combines monitoring, prediction and specification mining
- Uses at its core the notion of parametric specification, together with algorithms for parametric trace slicing used across any of the monitoring, prediction and mining capabilities
- RV shows that the three apparently different technologies, namely monitoring, prediction and mining, in fact belong together